Project 2: User Programs

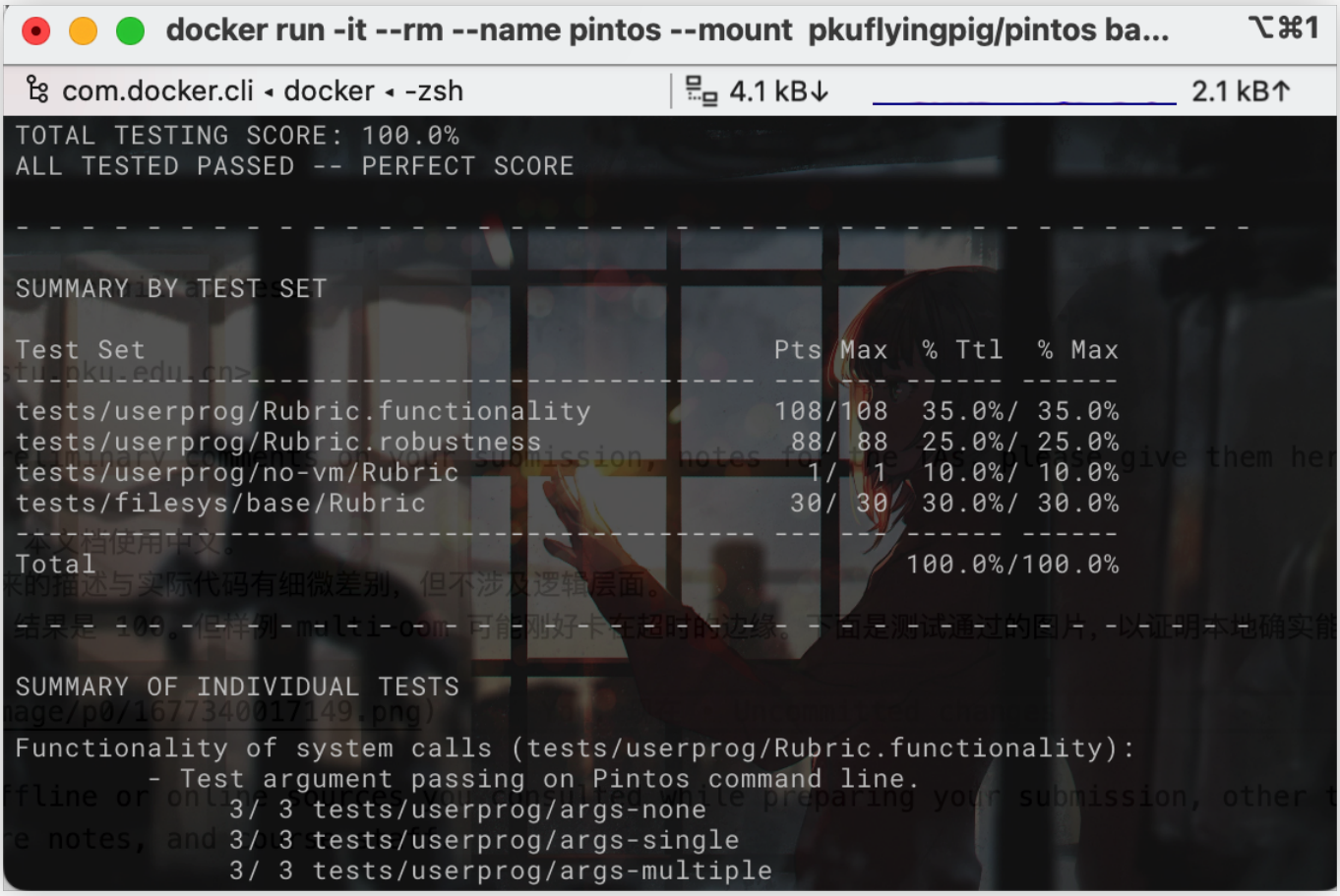
Preliminaries

> Fill in your name and email address.

寿晨宸 2100012945@stu.pku.edu.cn

> If you have any preliminary comments on your submission, notes for the TAs, please give them here.

1. 出于表述准确性考虑，本文档使用中文。
2. 为了方便理解，接下来的描述与实际代码有细微差别，但不涉及逻辑层面。
3. 本地 make grade，结果是 100。但样例 multi-oom 可能刚好卡在超时的边缘。下面是测试通过的图片，以证明本地确实能获得满分。



> Please cite any offline or online sources you consulted while preparing your submission, other than the Pintos documentation, course text, lecture notes, and course staff.

Argument Passing

DATA STRUCTURES

> A1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
/* 规定可以传入的参数的最大数量 */
#define MAX_ARG_NUM 128
/* 指针大小 */
#define PTR_SIZE (sizeof(void *))
```

ALGORITHMS

> A2: Briefly describe how you implemented argument parsing. How do you arrange for the elements of argv[] to be in the right order?
> How do you avoid overflowing the stack page?

How to implement argument parsing?

在参数解析的整个过程中，主要函数的调用顺序为：`process_execute()->thread_create()->start_process()->load()->process_pass_args()`，其中`process_pass_args()`为我添加的函数，主要用途是设置堆栈。

为了避免竞争，当我在在这一系列调用中传递字符串参数时，会将其分配到堆上。下文所有字符串都默认已经过拷贝，并不再赘述`malloc/free`的具体操作。

接下来是具体实现：

1. `process_execute(const char *cmd)`：调用`strtok_r()`，从 **cmd** 中提取出调用的可执行文件名 **file_name**，接下来调用`thread_create (file_name, PRI_DEFAULT, start_process, cmd)`，其中 **file_name** 将作为进程名，而 **cmd** 将作为传递给函数`start_process()`的参数。
2. `thread_create ()`：设置线程信息后，简单地调用`start_process(cmd)`。
3. `start_process(void *cmd)`：从 **cmd** 中提取出 **file_name**，调用`load(file_name,...)`，将可执行文件加载进内存中，并设置栈指针。然后调用`process_pass_args(cmd)`，将参数按规定的顺序压入栈中。
4. `process_pass_args(void **esp, void *cmd)`：
   - 将栈指针 **esp** 初始化为 user-virtual-memory 的最高点，即 **PHYS_BASE(0xc0000000)**
   - 循环调用`strtok_r()`将 **cmd** 拆分为各参数。
   - 将这些参数本身放在栈顶。（此时不考虑对齐）
   - 填充 **word_align**，使 **esp** 为 4 的倍数
   - 先推入`argv[argc]=NULL`，作为哨兵指针，然后按从右到左的次序，将各参数在栈顶的地址推入栈中。
   - 将 argv[0] 的地址和 argc 推入栈中。
   - 最后，将一个虚假的返回地址推入栈中，尽管 entry function 永远不会返回，但它应该与其他函数调用一样拥有相同的结构。

How do you arrange for the elements of argv[] to be in the right order?

按从左到右的顺序分解各参数，并将各参数的地址存入一个数组中。在推入参数的地址时，按从右到左的顺序访问该数组。

How do you avoid overflowing the stack page?

事实上，我并不检查每一次的 **esp** 值是否有效，不然的话效率太低了。我只是简单地遍历所有内容，若 **esp** 溢出或者访问失败，抛出 **page fault**，在处理 **page fault exception** 时令进程退出，并设置 **exit state** 为 -1。

RATIONALE

> A3: Why does Pintos implement strtok_r() but not strtok()?

`strtok` 是一个线程不安全的函数，因为它使用了静态分配的空间来存储被分割的字符串位置，而 `strtok_r()` 的占位符 **save_ptr** 是由调用者本身提供的。由于线程随时可能中断，所以应该使用 `strtok_r()`。

> A4: In Pintos, the kernel separates commands into a executable name and arguments. In Unix-like systems, the shell does this separation. Identify at least two advantages of the Unix approach.

1. 可以增加系统的鲁棒性。它可以帮助防止恶意程序通过修改参数来执行危险的操作。此外，分离参数还可以防止用户意外地在命令行中执行不安全的操作。譬如事先检查可执行文件是否存在。或者检察参数数量是否超过了限度。
2. 可以降低系统在内核中停留的时间。
3. 可以使对命令行的处理更加方便。例如，用户可以通过添加或删除参数来改变命令的行为，或者通过使用管道和重定向来将命令的输出发送到其他程序或文件中。
4. 更好的可移植性：将可执行文件名和参数分离可以使程序更加容易地在不同的操作系统和环境中移植。这是因为不同的操作系统和环境可能对参数的解析方式存在差异，但是可执行文件名通常是通用的。

System Calls

DATA STRUCTURES

> B1: Copy here the declaration of each new or changed struct or struct member, global or static variable, typedef, or enumeration. Identify the purpose of each in 25 words or less.

```
struct thread
{
   struct as_child_info *as_child;  /**< 记录当前进程作为子进程的信息. */
   struct list childs;              /**< 当前进程的子进程. */
   struct thread* parent;           /**< 父进程. */
   int exit_state;                  /**< 用于记录退出状态，初始化为 0. */
   bool success;                    /**< 记录当前进程是否被成功加载/运行. */
   struct semaphore sema_exec;      /**< 实现 exec 时的同步. */

   struct list files;               /**< 存储进程打开的文件. */

   int max_alloc_fd;                /**< 记录描述符池中已经被分配的最大文件描述符，用于分配下一个打开的文件描
   struct file* exec_prog;          /**< 记录当前进程正在运行的文件，该文件无法被修改. */
};
```

```
/**
 * 存储当前进程作为子进程的信息，因为父进程应当能在子进程消亡后仍能访问这些信息，
 * 所以不应该直接把这些信息写在 thread 里，而应另起结构体，并用 malloc
 * 将该结构体分配到堆上。
 */
struct as_child_info
{
   /* 初始化为本进程的 t_id，在进程消亡后父进程仍能访问之 */
   tid_t tid;
   /* 记录当前进程是否终止 */
   bool is_alive;
   /* 记录当前进程是否已经被 wait */
   bool is_waited;
   /* 在进程消亡后记录其 exit_state */
   int store_exit_state;
   /* 指向进程本身，当进程消亡后被置为 NULL */
   struct thread *process_self;
   /* 信号量，用于实现 wait 时的同步 */
   struct semaphore wait_sema;
   /* 作为子线程，供父进程访问的抓手 */
   struct list_elem as_child_elem;
};
```

```
/* 用于存储进程打开的文件的信息 */
struct thread_file
{
   /* 文件描述符 */
   int fd;
   /* 文件指针 */
   struct file* file;
   struct list_elem file_elem;

};
```

> B2: Describe how file descriptors are associated with open files. Are file descriptors unique within the entire OS or just within a single process?

进程会按照递增的顺序给文件分配文件描述符。具体流程如下：

我们使用 **max_malloc_fd** 存储已经被分配过的文件描述符中最大的，该成员变量被初始化为 **STDOUT_FILENO(1)**。
每当进程打开一个新的文件，进程会给该文件在堆上分配一块专属的空间，用于存储打开的文件的信息。并将文件的文件描述符设置为 **max_malloc_fd+1**，然后令 `max_malloc_fd++` 。

文件描述符仅对于单个进程是独一无二的。但对整个操作系统而言是可重复的。

ALGORITHMS

> B3: Describe your code for reading and writing user data from the kernel.

Read:

- 检查指向参数的指针是否合法。若合法，将其内容存入 **fd**，**buffer**，**size**，在接下来的操作中，我们将从文件 **fd** 中读 **size** 字节的信息，并写入 **buffer**。若非法，直接调用 exit(-1)。

- 判断 **fd**：
  - 若 `fd==STDOUT_FILENO`，即从标准输出读入内容，我们将其看作未定义行为，直接终止进程。
  - 若 `fd==STDIN_FILENO`，即从标准输入读入内容，这是合法的。我们调用 `input_getc()` 从标准输入逐字节地读取内容，并将之写入 **buffer**，最后返回 **size**。
  - 否则，检查 **fd** 是否在当前线程打开的文件列表中。
    - 若不在，说明当前进程无权访问 **fd** 或文件不存在，直接返回 -1。
    - 若存在，则先通过对文件系统加锁保护并发安全，然后调用 `file_read()`，将返回值设为 `file_read()` 的返回值。最后释放对文件系统的锁。

Write:

- 检查指向参数的指针是否合法。若合法，将其内容存入 **fd**，**buffer**，**size**，在接下来的操作中，我们将从 **buffer** 中读 **size** 字节的信息，并写入文件 **fd**。若非法，直接调用 exit(-1)。
- 判断 **fd**：
  - 若 `fd==STDIN_FILENO`，即向标准输入写内容，我们将其看作未定义行为，直接终止进程。
  - 若 `fd==STDOUT_FILENO`，即向标准输出写内容，这是合法的。我们调用 `putbuf(buffer, size)` 一次性地将内容写入标准输出，并返回 **size**。
  - 否则，检查 **fd** 是否在当前线程打开的文件列表中。
    - 若不在，说明当前进程无权访问 **fd** 或文件不存在，直接返回 -1。
    - 若存在，则先通过对文件系统加锁保护并发安全，然后调用 `file_write())`，将返回值设为 `file_write()` 的返回值。最后释放对文件系统的锁。

> B4: Suppose a system call causes a full page (4,096 bytes) of data
> to be copied from user space into the kernel. What is the least
> and the greatest possible number of inspections of the page table
> (e.g. calls to pagedir_get_page()) that might result? What about
> for a system call that only copies 2 bytes of data? Is there room
> for improvement in these numbers, and how much?

A full page：

- 若这一整页是连续的，最少需要 1 次。即第一次检查时发现其开始地址正好在页头，就不必再检查结束地址了。
- 若这 4096 bytes 是连续的，最多需要 2 次，即检查其开始地址和结束地址。
- 若这 4096 bytes 是离散的，最多需要 4096 次，因为每一 byte 的地址都需要被检查。

Two bytes：

- 最少 1 次。如果在检查开始地址时发现其距离页尾至少有 2 bytes 的空间，就不必再检查结束地址了。
- 最多 2 次。需要依次检查 2 bytes 的地址。

我暂时没有想出来有什么改进的办法。

> B5: Briefly describe your implementation of the “wait” system call
> and how it interacts with process termination.

我主要在函数 `process_wait()` 中实现系统调用 “wait”，并通过 `thread_exit()` 实现一些交互。接下来我们用 P 代表父进程，用 C 代表子进程。

P **Wait**： 遍历子进程列表 **P->childs**，搜索要等待的进程 C。

- 若 C 不是 P 的子进程，直接返回 -1。
- 若 C 是 P 的子进程。
  - 若 C 正在被等待（通过访问 **C->as_child->is_waited**）。直接返回 -1。
  - 若 C 已经消亡（通过 **C->as_child->is_alive** 判断），直接返回 C 的退出状态（通过访问 **C->as_child->store_exit_state**）。
  - 若 C 仍在运行，则将 C 标记为正在被等待。通过调用 `sema_down(&C->as_child->wait_sema)` 等待 C 消亡，在 C 消亡后，返回 C 的退出状态。

C **Terminate**：

- 将 **C->as_child->is_alive** 设置为 false。说明 C 已经消亡。
- 将 **C->exit_state** 存入 **C->as_child->store_exit_state**，让 C 的退出状态可以被 P 访问。
- 若 C 正在被等待，则通过调用 `sema_up(&C->as_child->wait_sema)`，将控制权交给 P。

> B6: Any access to user program memory at a user-specified address can fail due to a bad pointer value. Such accesses must cause the process to be terminated. System calls are fraught with such accesses, e.g. a “write” system call requires reading the system call number from the user stack, then each of the call’s three arguments, then an arbitrary amount of user memory, and any of these can fail at any point. This poses a design and error-handling problem: how do you best avoid obscuring the primary function of code in a morass of error-handling? Furthermore, when an error is detected, how do you ensure that all temporarily allocated resources (locks, buffers, etc.) are freed? In a few paragraphs, describe the strategy or strategies you adopted for managing these issues. Give an example.

在用户申请内存访问时，我们仅检查指针是否在用户内存区域内，而不检查其是否属于当前进程的内存区域。若访问非法，触发 page fault 后，再处理这个异常。下面是具体内容：

在访问内存之前，我们先判断指针是否属于用户区域。若不属于，直接调用 exit(-1)。

我们因此实现了五个函数，用于检查指针。

```
/* 在用户虚拟地址 UADDR 读取一个字节。
   UADDR 必须低于 PHYS_BASE。
   如果成功则返回字节值，如果发生段错误则返回 -1。*/
static int get_user(const uint8_t *uadder);

/* 在用户虚拟地址 UDST 写 BYTE 个字节
   UDST 必须低于 PHYS_BASE.
   如果成功则返回 true，如果发生段错误则返回 false */
static bool put_user(uint8_t *udst,uint8_t byte);

/* 系统调用的指针非法时，终止当前进程，并设置 exit_state=-1*/
void terminate_offend_process(void);

/* 判断从指针 ptr 开始的 size 个字节是否能合法读取
   若是，返回 ptr，若不是，调用 terminate_offend_process()
   终止进程，无返回 */
static void *check_read_vaddr(const void*,size_t);

/* 判断从指针 ptr 开始的 size 个字节是否能合法写入
   若是，返回 ptr，若不是，调用 terminate_offend_process()
   终止进程，无返回 */
static void *check_write_vaddr(void *, size_t);

/* 由于字符串的 size 不是固定的，其长度由 '\0' 标识，所以需要特定的判断函数 */
static void *check_str_vaddr(const char* str);
```

以系统调用 **“read”** 为例，根据文档所规定的堆栈的规则，它在栈上放了 3 个参数和对应的参数地址，3 个参数分别是 **(int)fd**，**(char *)buffer**，**(unsigned)size**，3 个参数的地址分别为 **esp+PTR_SIZE**，**esp+2*PTR_SIZE**，**esp+3*PTR_SIZE**。

系统调用 **“read”** 从 **fd** 读入 **size** 字节并写入 **buffer**。

在实现 **“read”** 时，我们首先要使用 `check_read_vaddr()` 检查三个参数的地址，判断其是否可以合法读取。(譬如判断 **fd** 的地址是否合法，会调用 `check_read_vaddr(user_ptr + PTR_SIZE,sizeof(int))`)。若不可以合法读取，在检查的过程中进程就会退出，并将退出状态设置为 -1。若可以合法读取，则直接通过访问 `check_read_vaddr()` 的返回值读取参数。在读取的参数中，**buffer** 是特别的，因为我们要向 **buffer** 中写 **size** 字节的信息，所以我们要通过调用 `check_write_vaddr(buffer,size)` 来判断从地址 **buffer** 开始的 **size** 个字节能否合法写入。

如果错误依然发生了，将触发 page fault，那么我们将在 `page_fault()` 中处理它。

在内核上下文中发生 page fault 的唯一可能是在处理 system call 中用户提供的指针时发生了错误。所以我们需要判断该特殊情况，用 eip 保存 eax 并将 eax 设置为 -1。然后直接返回。

否则，若在用户态发生 page fault，会调用 `kill()` 终止进程，将进程的退出状态设置为 -1。

以样例 **userprog/bad_jump2** 为例，它访问了非法内存，且无法提前被检查（`*(int*)0xC0000000 = 42`），所以必然会触发 page fault。又由于它是在用户态访问的非法内存，所以会调用 `kill()` ，并触发 `exit(-1)` 。

由于进程所有释放资源的行为都在其退出时完成，所以若进程因访问了非法地址退出时，仍能保证释放了其持有的资源。

SYNCHRONIZATION

> B7: The “exec” system call returns -1 if loading the new executable fails, so it cannot return before the new executable has completed loading. How does your code ensure this? How is the load success/failure status passed back to the thread that calls “exec”?

给出父进程 P 和子进程 C。我们通过操作信号量 **P->exec_sema** (initial 0)来实现同步，使用 **P->success** (initial false)来记录 C 加载成功/失败。因为 C 随时可能退出，而且在创建过程中也缺少可以直接访问 C 的接口，所以我们选择将 C 加载的状态放在 P 中。

P 想要根据 **P->success** 决定 **exec** 的返回值是什么，若 C 成功加载，exec 将返回 C 的 tid，若失败，则将返回 -1。所以在 C 完成修改 **P->success** 之前，P 应该等待， P 会调用 `sema_down(&P->exec_sema)` 。然后在做出判断之后，将 **P->success** 复原为 false。

C 应当根据加载的成功与否，做出相应的操作。若 C 成功加载，它会设置堆栈，将 **C->parent->success** 即 **P->success** 置为 true。若C 加载失败，它将直接退出。无论加载成功与否，最后 C 都会通过 `sema_up(&C->parent->exec_sema)` 将控制权转交给父进程 P。

> B8: Consider parent process P with child process C. How do you ensure proper synchronization and avoid race conditions when P calls wait© before C exits? After C exits? How do you ensure that all resources are freed in each case? How about when P terminates without waiting, before C exits? After C exits? Are there any special cases?

P wait©: P 通过访问 **C->as_child->is_alive** 获知 C 是否已经退出。

- C 已经退出：直接返回 **C->as_child->store_exit_state**。
- C 未退出：通过信号量 **C->as_child->wait_sema** (初始化为 0) 实现同步。当 P 调用 wait© 时，P 使用 `sema_down()` 。当 C 退出时，调用 `sema_up()` ，这确保了 P 在 C 退出之前阻塞，wait© 返回 **C->as_child->store_exit_state**。
- C 已经被等待过：直接返回 -1。（访问 **C->as_child->is_waited**）
- C 不是 P 的子进程或 C 不存在：直接返回 -1。（通过遍历 P 的子进程列表，确认 C 是否为 P 的子进程）

如何确保所有资源都被释放?

当进程 T 退出时，它释放 **T->childs** 和 **T->files** 中的所有资源。并将子进程的父进程置为空（通过 **as_child->process_self**）。

- 安全性
  - 子进程列表：子进程列表只会被其父进程访问，而此时父进程已经消亡，资源可以释放。（子进程只会在其父进程仍存在时访问 **as_child**）
  - 文件列表：当打开文件的进程退出后，文件列表也不会被访问了。
- 完全性：
  - 子进程：它的资源被只被父进程分配，最后在父进程消亡后被统一释放。这是一一对应的。
  - 文件：它的资源在被进程打开时分配，在进程消亡时释放，是一一对应的。

P 退出:

- C 已经退出：直接释放 **C->as_child**。
- C 未退出：
  - 令 **C->parent=NULL**。
  - 释放 **C->as_child**，因为它不会再被其父进程访问了，而只有当其父进程仍存在（**C->parent!=NULL**）时，C 才会修改 **C->as_child** 中的元素。所以这种释放是安全的。

RATIONALE

> B9: Why did you choose to implement access to user memory from the
> kernel in the way that you did?

因为第二种方式更快，且更广泛地应用于真实的内核，而且文档也给予了充分的提示。所以我倾向于选择这种方式。

> B10: What advantages or disadvantages can you see to your design
> for file descriptors?

Advantages:

1. 更符合直觉。每个进程拥有独立的文件描述符池，而非所有进程共享一个统一的文件描述符池。提供了一种进程独自占有空间的假象。
2. 不会消耗太多的内核空间。不会因为进程打开了太多的文件导致内核崩溃。
3. 对于单个进程而言，能更加方便地访问文件。
4. 提供了对于文件系统的保护，不会因为一个进程崩溃导致整个文件系统出现问题。

Disadvantages:

1. 在用户虚拟空间中会额外分配很多冗余的资源用于存储文件信息。
2. 不方便内核访问所有打开的文件资源。

> B11: The default tid_t to pid_t mapping is the identity mapping.
> If you changed it, what advantages are there to your approach?

在我的实现中，我并没有改变这种映射。因为 Pintos 中进程与线程一一对应，并不存在多线程的进程，所以令 tid 与 pid 一一对应是很自然且高效的实现，没有必要改变它。